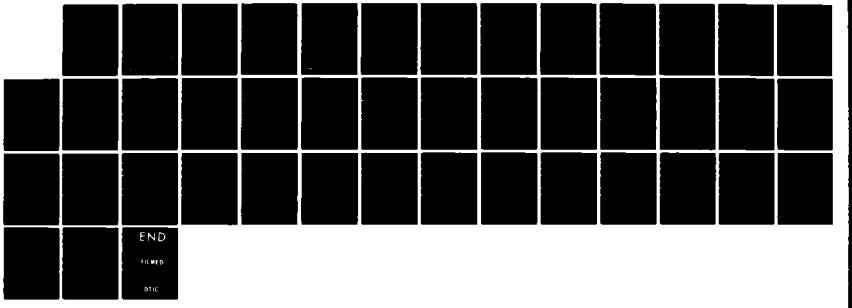
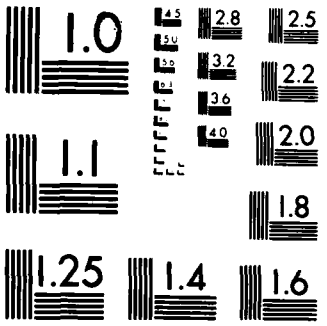
AD-A149 138 MONITORING THE EXECUTION OF PLANS IN SIPE (SYSTEM FOR INTERACTIVE PLANNIN..(U) SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTE.. D E WILKINS 1/1

UNCLASSIFIED 06 SEP 84 AFOSR-TR-84-1161 F49620-79-C-0188 F/G 15/7 NL





15

MONITORING THE EXECUTION OF PLANS IN SIPE

Final Technical Report

September 6, 1984

By: David E. Wilkins, Computer Scientist

Artificial Intelligence Center
Computer Science and Technology Division

Prepared for:

Air Force Office of Scientific Research
Building 410
Bolling Air Force Base
Washington, D.C. 20332

ATTENTION: Dr. Buchal

AFOSR Contract No. F49620-79-C-0188
SRI Project 8871

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-2046

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): SRI Project 8871	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 84-1161
6a. NAME OF PERFORMING ORGANIZATION: SRI, International; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code): Artificial Intelligence Ctr, Computer Science & Technology Div, 333 Ravenswood Avenue, Menlo Park CA 94025	7b. ADDRESS (City, State and ZIP Code): Directorate of Mathematical and Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49620-79-C-0188
8c. ADDRESS (City, State and ZIP Code): Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT NO
61102F	2304	A2	

11. TITLE (Include Security Classification): MONITORING THE EXECUTION OF PLANS IN SIPE

12. PERSONAL AUTHOR(S): David E. Wilkins

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM 1/9/83 TO 31/8/84	6 SEP 84	38

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In real-world domains (a mobile robot is used as a motivating example), things do not always proceed as planned. Therefore it is important to develop better execution-monitoring techniques and replanning capabilities. This paper describes the execution-monitoring and replanning capabilities of the SIPE planning system. (SIPE assumes that new information to the execution monitor is in the form of predicates, thus avoiding the difficult problem of how to generate these predicates from information provided by sensors.) The execution-monitoring module takes advantage of the rich structure of SIPE plans (including a description of the plan rationale), and is intimately connected with the planner, which can be called as a subroutine. The major advantages of embedding the replanner within the planning system itself are: (1) The replanning module can take advantage of the efficient frame reasoning mechanisms in SIPE to quickly discover problems and potential fixes; (2) The deductive capabilities of SIPE are used to provide a reasonable solution to the truth maintenance problem; and (3) The planner can be (CONTINUED)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED X SAME AS RPT ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Robert N. Buchal	22b. TELEPHONE NUMBER (Include Area Code): 202, 767- 4939
	22c. OFFICE SYMBOL: NM

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

ITEM #19, ABSTRACT, CONTINUED: called as a subroutine to solve problems after the replanning module has inserted new goals in the plan. Another important contribution is the development of a general set of replanning actions that will form the basis for a language capable of specifying error-recovery operators, and a general replanning capability that has been implemented using these actions.

SRI International

MONITORING THE EXECUTION OF PLANS IN SIPE

Final Technical Report

September 6, 1984

By: David E. Wilkins, Computer Scientist

Artificial Intelligence Center
Computer Science and Technology Division

Prepared for:

Air Force Office of Scientific Research
Building 410
Bolling Air Force Base
Washington, D.C. 20332

ATTENTION: Dr. Buchal

AFOSR Contract No. F49620-79-C-0188
SRI Project 8871

Approved:

Stanley J. Rosenschein, Director
Artificial Intelligence Center

Donald L. Nielson, Acting Director
Computer Science and Technology Division

SRI International®

333 Ravenswood Ave • Menlo Park, CA 94025
415 326-6200 • TWX 910-373-2046 • Telex 334-486

1. Introduction

Research on planning and problem-solving systems was begun at SRI International in September 1979 under AFOSR sponsorship (SRI Project 8871; Contract No. F49620-79-C-0188). Progress has been described in detail in four annual reports (1980, 1981, 1982, and 1983). This report describes the research performed during the past year under AFOSR Contract F49620-79-C-0188. The research peformed during the first four years of the project is described in a paper which appeared in the April 1984 issue of the Artificial Intelligence Journal [4].

The main task of this research program is to develop powerful methods of representing, generating, and executing hierarchical plans that contain parallel actions. Execution involves monitoring the state of the world and, possibly, replanning if things do not proceed as expected. Two different approaches to these problems are being pursued under this contract. The first is heuristic; it involves building an actual computer program that provides a representation from which it then generates plans. During the past year, the vast majority of the effort on this project has been concentrated on this effort. In particular, execution-monitoring and replanning capabilities have been developed. The second approach is to investigate the theoretical foundations of planning. This has not resulted in a program, but has helped formalize the planning problem and one solution to it.

This report briefly summarizes the research performed in both these areas. Two papers are enclosed, which provide detailed descriptions of what has been accomplished.

2. Execution Monitoring in SIPE

A principal goal of our research in planning and plan execution is the development of a heuristic system that can plan an activity and then monitor the execution of that plan. While logical formalisms seem advantageous for certain types of reasoning (e.g., metaplanning), most approaches based on logic still suffer from inefficiency because of an inability to control the possible deductions. A number of researchers (here at SRI, at Stanford, and at other centers) are exploring

such approaches, while the heuristic approach used in SIPE is unique and promising.

Over the last few years we have designed and implemented such a system, SIPE, (System for Interactive Planning and Execution Monitoring) [4]. The basic approach to planning is to work within the hierarchical-planning paradigm, representing plans in procedural networks – as has been done in NOAH [2] and other systems. Several extensions of previous planning systems have been implemented, including the development of a perspicuous formalism for describing operators and objects, the use of constraints for the partial description of objects, the creation of mechanisms that permit concurrent exploration of alternative plans, the incorporation of heuristics for reasoning about resources, and the creation of mechanisms that make it possible to perform deductions.

This year we have begun using the planning of tasks for a mobile robot as a motivating domain. This has led to certain additions to the SIPE planning system. During the past year we have implemented conditional plans within SIPE, which may cause the plan to wait for information-gathering actions to be executed. We have also added the ability to represent some types of uncertainty by permitting predicates and certain variables to be unknown. The deductive capability of SIPE has been expanded to be more powerful and to handle these unknowns. These features are described in more detail in the enclosed paper by David Wilkins. SIPE also contains "hooks" for incorporating special-purpose subsystems for geometric modeling or spatial reasoning, things we do not intend to investigate.

During the past year, we have implemented an execution-monitoring and replanning capability within the SIPE planning system. This is described in detail in the enclosed paper by David Wilkins, which is being submitted to the Computational Intelligence Journal. The main features are briefly summarized here.

Given correct information about unexpected events, SIPE is able to determine how this affects the plan being executed, and, in many cases, is able to retain most of the original plan by making changes in it to avoid problems caused by these unexpected events. It also is capable of shortening the original plan when serendipitous events occur. It cannot solve difficult problems involving drastic changes to the expected state of the world, but it does handle many types of small errors that may happen frequently in a mobile robot domain. The execution-monitoring package does

this without the necessity of planning originally to check for these errors.

The major contributions of this work center around taking advantage of the rich structure of SIPE's planner and its plans, and the development of a general set of replanning actions that are used as the basis of an automatic replanner and can be used as the basis of a language for specifying domain-dependent error-recovery information. The replanner calls the standard planning system so that it can take advantage of the efficient frame reasoning mechanisms in SIPE to quickly discover problems and potential fixes, and use the deductive capabilities to provide a reasonable solution to the truth maintenance problem. The fixes need involve only inserting new goals in the plan, since calling the planner as a subroutine will solve these goals in a manner that assures there will be no conflicts with the rest of the plan. SIPE's execution-monitoring capabilities make extensive use of the explicit representation of plan rationale in plans. The problem detector does not remove parts of the original plan unless the parts are actually problematical. SIPE's deductive capability is instrumental in the solution of the truth maintenance problem.

Another important contribution is the development of a general set of six replanning actions, REINSTANTIATE, INSERT, INSERT-CONDITIONAL, RETRY, REDO, and POP-REDO. These have all been implemented and are described in detail in the enclosed paper. They will form the basis for a language capable of specifying error-recovery operators, and a general replanning capability, which has been implemented. These actions provide sufficient power to alter plans in a way that often retains much of the original plan, (e.g., the REINSTANTIATE action). The general replanner attempts to solve all problems that are found.

The major limitations of this research result from the assumption of correct information about unexpected events. This avoids the hard problems of generating predicates from information provided by the sensors, deciding how much effort to expend checking facts that may be suspect, and modeling uncertain or unreliable sensors. These problems are all crucial to providing execution-monitoring capabilities to a mobile robot, and we hope to address these problems in the future.

3. Theoretical Foundations

Like most planning systems, SIPE assumes discrete states and makes no provision for more general statements about what happens during an action. During the past year, we have studied this problem to determine what general statements one would like to make about what happens during the performance of actions. This has led to the development of a device called a process model, which is used to represent the observable behavior of an agent in performing an action. This model is more general than previous models of action, allowing sequencing, selection, nondeterminism, iteration, and parallelism to be represented. This is described in more detail in the enclosed paper by Michael Georgeff [1], which describes work that was partially supported by this contract.

Theoretical work supported by this project has also looked at the problem of rationality. Most work on robot planning and problem-solving is done against the background of an implicit, but unarticulated theory of rational action. Roughly stated, this theory assumes a rational agent, who attempts to maintain a state of consistency between his intentions (plans) and his beliefs and goals; the agent will perform (and intend to perform) those actions he believes will achieve his goals. We have developed a model of the cognitive agent in which the principle of rationality is formalized and studied abstractly. The model has immediate value as an analytical tool and a way of integrating such topics as belief revision, execution monitoring, replanning, and various types of goals (e.g., maintenance and prevention) in a common theoretical framework.

During this past year work has focused on several related issues. First, we have looked at the intention component of our model more carefully, attempting to formalize suitable semantic conditions on the content of intentions (e.g., that they describe states of affairs the agent believes he can bring about). Second, we have looked at alternatives to the syntactic realization of beliefs as expressions in an internal language so that the closure of beliefs under logical consequence might be achieved without requiring explicit syntactic deduction. Third, we have made progress in formalizing the connection between perception and knowledge and are beginning to develop logical tools for reasoning about the information content of percepts.

4. Publications

A paper entitled "Domain-independent Planning: Representation and Plan Generation", appeared in the Artificial Intelligence Journal in the April 1984 issue [4]. This paper describes all work on SIPE before the development of the execution monitoring package. A condensed version of this paper appeared as a long paper in the 1983 Proceedings of the International Joint Conference on Artificial Intelligence. This project supported the presentation of this paper by David Wilkins at the conference in Karlsruhe, Germany in August 1983.

This project partially supported the research described in "A Theory of Action for MultiAgent Planning" by Michael Georgeff, which appeared in the proceedings of the 1984 AAAI Conference in Austin, Texas. In addition, the enclosed paper, entitled "Monitoring the Execution of Plans in SIPE", is being submitted to the Computational Intelligence Journal.

REFERENCES

1. Georgeff, M., "A Theory of Action for MultiAgent Planning", *Proceedings 1984 AAAI Conference*, Austin, Texas, 1984, pp. 121-125.

2. Rosenschein, S., "Plan Synthesis: A Logical Perspective", *Proceedings IJCAI-81*, Vancouver, British Columbia, 1981, pp. 331-337.

3. Sacerdoti, E., *A Structure for Plans and Behavior*, Elsevier, North-Holland, New York, New York, 1977.

4. Wilkins, D., "Domain-independent Planning: Representation and Plan Generation", *Artificial Intelligence 22*, April 1984, pp. 269-301.

Monitoring the Execution of Plans in SIPE

By

David E. Wilkins

Artificial Intelligence Center

SRI International

ABSTRACT

In real-world domains (a mobile robot is used as a motivating example), things do not always proceed as planned. Therefore it is important to develop better execution-monitoring techniques and replanning capabilities. This paper describes the execution-monitoring and replanning capabilities of the SIPE planning system. (SIPE assumes that new information to the execution monitor is in the form of predicates, thus avoiding the difficult problem of how to generate these predicates from information provided by sensors.) The execution-monitoring module takes advantage of the rich structure of SIPE plans (including a description of the plan rationale), and is intimately connected with the planner, which can be called as a subroutine. The major advantages of embedding the replanner within the planning system itself are: 1) The replanning module can take advantage of the efficient frame reasoning mechanisms in SIPE to quickly discover problems and potential fixes, 2) The deductive capabilities of SIPE are used to provide a reasonable solution to the truth maintenance problem, and 3) The planner can be called as a subroutine to solve problems after the replanning module has inserted new goals in the plan. Another important contribution is the development of a general set of replanning actions that will form the basis for a language capable of specifying error-recovery operators, and a general replanning capability that has been implemented using these actions.

1. Introduction

A principal goal of our research in planning and plan execution is the development of a domain-independent, heuristic system that can plan an activity and then monitor the execution of that plan. Over the last two years we have designed and implemented such a system, SIPE (System for Interactive Planning and Execution Monitoring).[1] The basic approach to planning is to work within the hierarchical-planning paradigm, representing plans in procedural networks - as has been done in *NOAH* [2] and other systems. Several extensions of previous planning systems have been implemented, including the development of a perspicuous formalism for describing operators and objects, the use of constraints for the partial description of objects, the creation of mechanisms that permit concurrent exploration of alternative plans, the incorporation of heuristics for reasoning about resources, and the creation of mechanisms that make it possible to perform deductions.

Given a description of the world, and a set of operators that it can apply, SIPE can generate a plan to achieve a goal in the given world. However, in real-world domains, things do not always proceed as planned. Therefore, it is desirable to develop better execution-monitoring techniques and better capabilities to replan when things do not go as expected. In complex domains it becomes increasingly important to use as much as possible of the old plan, rather than to start all over when things go wrong.

This paper describes the execution-monitoring and replanning abilities that have recently been incorporated into the SIPE system. The particular advantages than can be obtained by the use of the rich structure in our plan representation are shown, as well as more general problems. The environment of a mobile robot has been used as a motivating domain in the development of some of the abilities here, though implementation has been in a general, domain-independent manner. This document does not describe resources, constraints, plan generation, and other features of SIPE, nor does it attempt to justify the basic assumptions behind the system. The interested reader is referred to [5] for this.

The problem we are addressing is the following: given a plan, a world description, and some appropriate description of an unexpected situation that occurs during execution of the plan,

[1]The research reported here is supported by Air Force Office of Scientific Research Contract F49620-79-C-0188.

transform the plan, retaining as much of the old plan as is reasonable, into a plan that will still accomplish the original goal from the current situation. This process can be divided into four steps - 1) discovering or inputting the information about the current situation, 2) determining the problems this causes in the plan, if any, (similarly, determining shortcuts that could be taken in the plan after unexpected but helpful events), 3) creating "fixes" that change the old plan, possibly by deleting part of it and inserting some newly created subplan, and 4) determining whether any changes made by the above fixes will conflict with remaining parts of the old plan. Steps 2 and 4, and possibly 3, involve solving a truth maintenance problem since it will be necessary to determine which aspects of a situation are necessary for later parts of the plan. In SIPE, step 4 becomes part of step 3 as only fixes that are guaranteed to work are produced. In addition, serendipitous effects are used to shorten the original plan in certain cases.

The major contributions of the execution-monitoring and replanning module in SIPE result from taking advantage of the system's rich plan representation and from imbedding it within the planning system itself, rather than implementing it as an independent module. This provides a number of advantages, of which the most important follow: 1) the replanning module can take advantage of the efficient frame reasoning mechanisms in SIPE to quickly discover problems and potential fixes, 2) the deductive capabilities of SIPE are used to provide a reasonable solution to the truth maintenance problem, and 3) the planner can be called as a subroutine to solve problems after the replanning module has inserted new goals in the plan. Another important contribution is the development of a general set of replanning actions that will form the basis for a language capable of specifying error-recovery operators (see Sections 5 and 6). A general replanning capability has been implemented using these actions.

SIPE assumes that information provided about unexpected events is correct and, to a certain extent, complete. This assumption avoids many of the hardest problems involved in getting a planner such as SIPE to control a mobile robot. The difficult problem of how to generate correct predicates from information provided by the sensors is not addressed. We expect the translation of the information from the robot's sensors (e.g., the pixels from the camera or the range information from the ultrasound) into the higher-level predicates used by the planner to be crucial to the

application of a SIPE-like planner to a mobile robot. We hope to address this problem in the near future.

In a mobile robot domain, it may often be important to spend considerable effort in checking for other things that might have gone wrong in addition to the unexpected occurrence already noticed. There is a large tradeoff here as interpreting visual input of unexpected scenes may be expensive. The research described here also does not address this problem as it assumes that the minimum is wrong in accordance with the information that has been given (after taking deductive operators into account). The problem of uncertain or unreliable sensors or information is largely unaddressed, except that some predicates and variables may be specified as unknown. What is discussed here is what to do with new information in the form of predicates (assuming you have somehow discovered such predicates). Replanning appropriately with such information is a necessary part of the overall solution.

Section 2 of this paper describes how plans are represented in SIPE. To describe unexpected situations, a user (currently a human, but eventually this may be a program controlling and interpreting the robot's sensors) can enter arbitrary predicates at any point in the execution or can specify certain things as unknown. Section 3 describes the details of this process. Once the description of the unexpected situation has been accumulated, the execution monitor calls a problem recognizer, described in Section 4, which returns a list of all the problems it detects in the plan.

In general, recovering from an arbitrary error is a very hard problem. Often very little of the existing plan can be reused. One can always fall back on solving the original problem in the new situation, ignoring the plan that was being executed. The replanning part of SIPE, however, tries to change the old plan, keeping as much of it as possible. Since the general problem is so difficult, one would not expect very impressive performance from a replanner that did not have domain-specific information for dealing with errors. For this reason, we have implemented a number of general replanning actions (i.e., actions that modify a plan in ways that are useful for handling unexpected situations) in SIPE that can be referenced in a language for providing domain-specific error-recovery instructions. In many domains, the types of errors that are commonly encountered

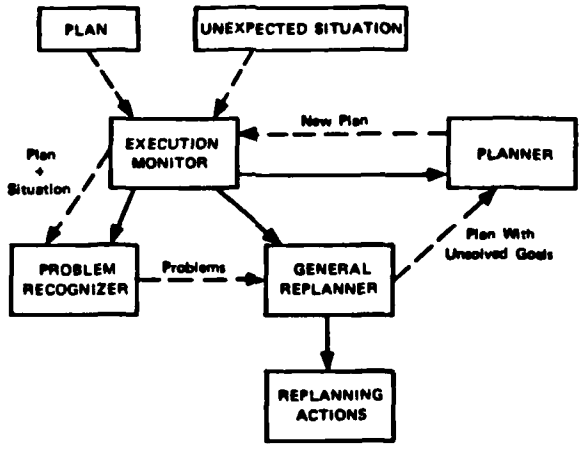


Figure 1
Control and Data Flow in SIPE Modules

can be predicted (e.g., the robot arm dropping something it was holding, or missing something it was trying to grasp). The user can then specify error-recovery operators for these errors, using SIPE'S replanning actions, to take appropriate action after expected errors.

In addition, SIPE provides a general replanning ability that can be applied in the general case and when there are no specific instructions. It is given the list of problems found by the problem recognizer, and tries certain replanning actions in various cases, but will not always find a solution. The replanning actions are described in Section 5 and the general replanner in Section 6. The general replanner changes the plan so that it will look like an unsolved problem to the standard planner in SIPE (e.g., by inserting new goals). After the replanner has dealt with all the problems that were found, then the planner is called on the plan (which now includes unsolved goals) and if it produces a new plan, this new plan should correctly solve all the problems that were found. Section 7 shows examples of the general replanner in operation.

Figure 1 shows the various modules in the SIPE execution-monitoring system. The solid arrows show which modules call which others. The broken arrows show the flow of data and information through the system as it replans for an unexpected situation. These arrows are labeled with a description of the data being passed.

2. Plans in SIPE

Plans in SIPE are represented as procedural networks [2] with temporal information encoded in the predecessor and successor links between nodes. The plan rationale is of primary importance to the execution monitor and is encoded in the network by MAINSTEP links between nodes, and by the use of PRECONDITION nodes (described below). MAINSTEP links describe how long each condition that has been achieved must be maintained. A context must also be given to completely specify a plan, as the network contains choice points from which alternative plans branch. The types of nodes that occur in plans are described below to the extent necessary for understanding the execution-monitoring capabilities.

SPLIT and JOIN nodes provide for parallel actions. SPLITs have multiple successors and JOINs have multiple predecessors so that partially ordered plans can be produced. JOIN nodes have a *Parallel-Postcondition* slot, which specifies the predicates that must all be true in the situation represented by the JOIN node. If a JOIN node originally has N predecessors, then there will be N conjunctions of predicates that must all be true at the JOIN node. (Some branches may have been linearized so there may be fewer than N predecessors after planning.) It is easier to record this at the JOIN node (than by having previous nodes point to the JOIN as their purpose), since a failed parallel-postcondition can more easily be retried during execution monitoring if there is easy access to all parallel-postconditions. The Parallel-Postcondition slot is filled only when the JOIN is first introduced into the plan – it is not updated as more detailed levels of the hierarchy are expanded. As long as the highest level predicates are as desired, it is assumed that the lower level predicates are irrelevant.

COND, ENDCOND, and CONDPATTERN nodes implement conditional plans. COND and ENDCOND are similar to SPLIT and JOIN, but each successor or the COND begins with a CONDPATTERN node that determines which successor will be executed.

CHOICE nodes denote branching points in the search space. They have multiple successors, but the context selects one of these as being in the current plan. Constraints on variables may be posted relative to this choicepoint. Thus, if the part of a plan after a CHOICE node is removed, the corresponding choicepoint in the context should also be removed from the context so that

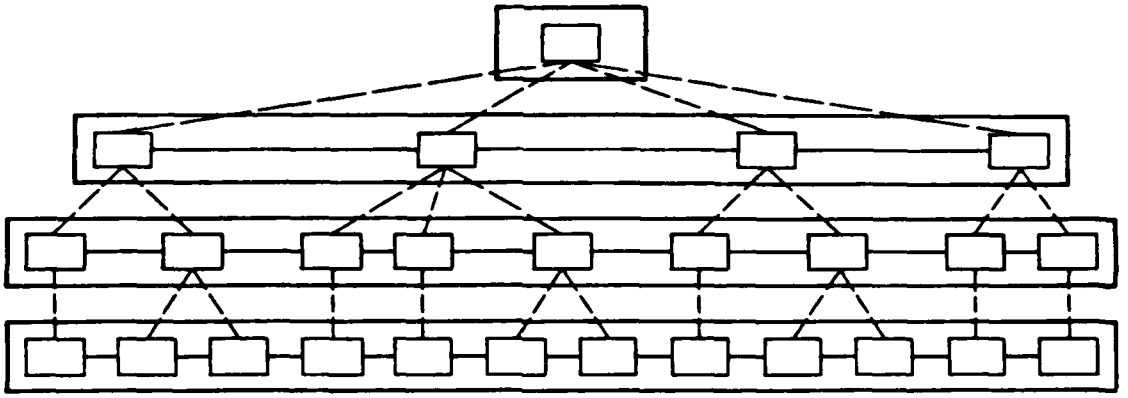
constraints that are no longer valid will be ignored.

GOAL nodes do not occur in final plans as they represent open problems that have not been solved yet. A GOAL node specifies a predicate that is a goal to achieve but which is not true in the situation represented by its location in the procedural network. Replanning actions will insert GOAL nodes in the plan. Each GOAL node has a MAINSTEP slot, which denotes a point later in the plan that depends on the GOAL. (This describes the rationale for having the GOAL in the plan.) Each goal must be maintained as true until the node which is its MAINSTEP is executed. A MAINSTEP slot can have the atom PURPOSE as its value, denoting that the given predicate is the main purpose of the plan, and not preparation for some later action.

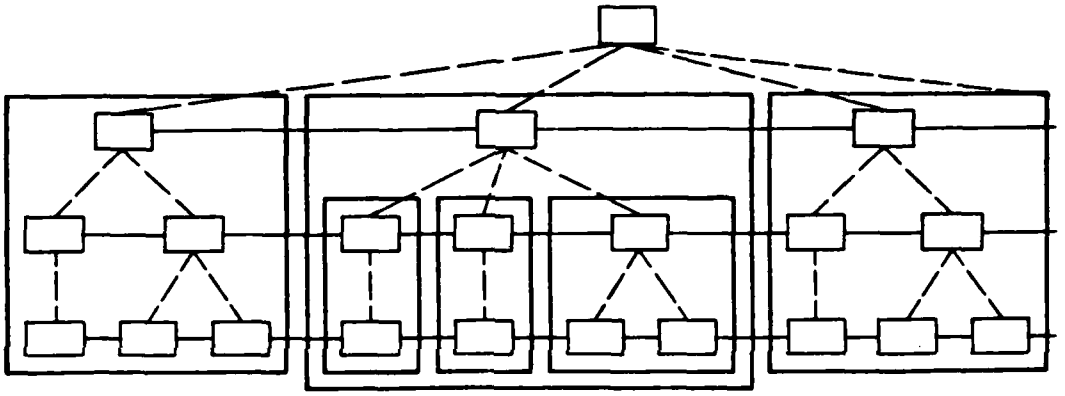
PHANTOM nodes are similar to GOAL nodes except that they are already true in the situation represented by their location in the procedural network. They are part of the plan because their truth must be monitored as the plan is being executed. They also contain MAINSTEP slots.

PROCESS nodes represent actions to be performed during execution of the plan, and also have MAINSTEP slots as do PHANTOM and GOAL nodes. In a final plan, all PROCESS nodes will denote primitive actions. (There are also CHOICEPROCESS nodes, which are like PROCESS nodes except that they have a list of actions, one of which must be performed.)

PRECONDITION nodes provide a list of predicates that must be true in the situation represented by their location in the procedural network. Operators may specify preconditions that must obtain in the world state before the operator can be applied. The concept of precondition here differs from its counterpart in some planners, since the system will make no effort to make the precondition true. A false precondition simply means that the operator is not appropriate. Conditions that the planner should make true (and therefore backward chain on) can be expressed as goal or process nodes. This effectively encodes metaknowledge about how to achieve goals as SIPE will use any means to solve a goal node, only the operators listed to solve a process node, and no operators will be tried to solve a PRECONDITION node. Thus, a precondition becoming false does not mean that it should be made into a goal; rather it means that the whole part of the plan produced by the operator producing this precondition is invalid. Such nodes help encode the rationale of a plan since they effectively mean that the part of the plan associated with them (see



(a) plans at different levels



(b) wedges used by the execution monitor

Figure 2
SIPE Plan Viewed from Different Perspectives

below) was produced on the assumption that the predicates in the precondition are true.

In addition to the "horizontal" MAINSTEP, predecessor, and successor links within one level of a plan, there are "vertical" links between different levels of the hierarchy. Each node that is expanded by the application of an operator has descendant links to each node so produced. The descendant nodes in turn have ancestor links back to the original node one level higher in the hierarchy. Starting with a node that was expanded by an operator application, a *wedge* of the plan is determined by following all its descendant links (in the current context) repeatedly (i.e., including descendants of descendants, etc.) to the lowest level. (This definition of wedges is the

same as that used by Sacerdoti in [2].) Figure 2 depicts this graphically, with the large boxes in part (b) representing wedges. The node originally expanded by an operator application is called the *top* of the wedge. A wedge with its top at a high level in the hierarchy will generally contain many lower level wedges within it, and the only nodes that can be the tops of wedges are GOAL, PROCESS, and CHOICEPROCESS nodes.

Since PRECONDITION nodes are created only when an operator is applied, the part of a plan associated with a PRECONDITION node can be found by traversing up the ancestor links to the point where the precondition first became part of the plan (once inserted, PRECONDITION nodes are copied down from level to level). The node that was expanded by an operator to create this precondition is one level higher than where the first PRECONDITION node appears and is the top of the wedge associated with each of the PRECONDITION nodes that are copied from this first one.

3. The Input of Unexpected Situations

During execution of a plan in SIPE, some person or computer system monitoring the execution can specify what actions have been performed and what changes have occurred in the domain being modeled. SIPE permanently changes its original world model to show the effects of actions that have already been performed. At any point during execution, the system will accept two types of information about the domain: 1) an arbitrary predicate whose arguments are ground instances that is now true, false or unknown, and 2) a local variable name that is now unknown. SIPE first checks whether the truth values for the new predicates are different from its expectations, and, if they are, it applies its deductive operators to them to deduce more changed predicates.

It is important to note that the inputting of predicates does not solve the "pixels to predicates" problem, which is the crucial problem in using a planner such as SIPE to control the actions of a robot. This problem involves translating the input of the robot's sensors (e.g., the pixels from the camera or the range information from the ultrasound) into the higher level predicates used by the planner. The research described here involves what to do with the predicates once they have been determined but does not address the question of how to determine them automatically. We hope

to address this latter problem in the near future.

3.1 Unknowns

Unknowns are a new addition to SIPE as it previously assumed complete knowledge of the world. Having unknown quantities constitutes a fundamental modification since even the method of determining whether a predicate is true must be changed. If the truth values of critical predicates are unknown, the planner will quickly fail since none of the operators will be applicable. (Neither a negated or an unnegated predicate in a precondition will match an unknown predicate.) Operators can require predicates to be unknown as part of their precondition, in case there are appropriate actions to take when things are uncertain. Conditional plans have also been implemented as part of the execution-monitoring package in SIPE, so an operator might produce a plan with an action to perceive the unknown value, followed by a conditional plan that specifies the correct course of action for each possible outcome of the perception action. The deductive capabilities have also been enhanced so that operators can deduce that something is unknown.

The ability to specify variables as unknown is simply a tool provided by the system that will hopefully be useful in some domains, and particularly in a mobile robot domain. The idea behind this tool is that the location of an object may become unknown during execution. Rather than make predicates unknown, which may cause the application of operators to fail, we simply say that the variable representing the location is instantiated to the atom UNKNOWN rather than to its original location. All predicates with have this variable as an argument may then still match as if they were true. Thus the system can continue planning as if the location were known. The only restriction is that no action can be executed that uses an unknown variable as an argument. When such an action is to be executed (e.g., go to LOCATION1) then the actual instantiation of the variable must be determined before executing the action (possibly through a perception action). Note that it would be incorrect to continue planning if the truth values of important predicates depended on the instantiation of the location variable. It is the responsibility of the user not to use this tool (i.e., the unknown variable) if predicates depend on the variable's value. This tool may or may not prove useful in practice.

3.2 Interpreting the input

SIPE assumes that the minimum is wrong in accordance with the information that has been given (after taking deductive operators into account). Alternatively, we could decide on some basis (which would have to be provided as part of the domain-specific description) how much effort to spend with perception actions to see if more than the minimum has gone wrong. For example, if we are told that (ON A B) is not true when we expected it to be, we might want to look if B is where we thought it was. As it is, SIPE will just deduce that B is clear (if no other block is on B) and will not try to execute actions to make further checks about the world. Doing the latter could be very expensive for a mobile robot without good domain-specific knowledge about what was worth checking.

The user need not report all predicates that have changed since many of these may be deduced by SIPE's deductive operators. The system's deductive power has been increased recently (see next section) so many effects can be deduced from certain critical predicates. There is a problem in deciding how the unexpected effects interact with the effects of the action that was currently being executed (e.g., did they happen before, during, or after the expected effects?). Our solution to this problem is to assume the action took place as expected and to simply insert a "Mother Nature" action after it that is assumed to bring about the unexpected effects (and things deduced from them). The system assumes that any effects of the action being executed that did not actually take place are either provided or can be deduced from the information that is provided. This solution interfaces cleanly and elegantly with the rest of the planner and avoids having to model how the unexpected effects might interact with the expected effects.

4. Finding Problems in a Plan

Having just inserted a MOTHER-NATURE node (MN node) in a plan being executed, SIPE must now determine how the effects of this node affect the remainder of the plan. This involves solving the truth maintenance problem, since it is necessary to know on which facts the remainder of the plan depended. This is discussed later in this section. Because of the rich information content in the plan representation (including the plan rationale), there are only six problems that

must be checked. These are discussed below. All problems in the remainder of the plan are found, and this list is later given to the general replanner, which attempts to change the plan to solve these problems.

1 – *Purpose not achieved.* If the MN node negates any of the main effects of the action just executed, then there is a problem. The main effects must be reachieved.

2 – *Previous phantoms not maintained.* SIPE keeps a list of phantom nodes that occur before the current execution point, and whose MAINSTEP slot specifies a point in the plan that has not been executed yet. These are phantoms that must be maintained. If the MN node negates any of these, then there is a problem. The phantoms that are no longer true must be reachieved.

3 – *Process node using unknown variable as argument.* If a variable has been declared as unknown, then the first action using it as an argument must be preceded by a perception action for determining the value of the variable (see Section 3).

4 – *Future phantoms no longer true.* A phantom node after the current execution point may no longer be true. It must be changed to a GOAL node so that the planner will try to achieve it.

5 – *Future precondition no longer true.* A PRECONDITION node after the current execution point may no longer be true. In this case, we do not want to reachieve it, but rather pop up the hierarchy and perform some alternative action to achieve the goal at that level of the hierarchy.

6 – *Parallel-postcondition not true.* All the parallel-postconditions may no longer be true at a JOIN node. (This could be handled by maintaining phantoms, but is more convenient to handle separately.) In this case, we must insert a set of parallel goals after the JOIN, one for each untrue parallel-postcondition. The parallel-postconditions of the new JOIN will be the same as those on the old JOIN.

Note that only the last three problems below require truth maintenance since only they must know the truth value of predicates in situations after the current execution point. In addition to the above problems, possible serendipitous effects are also noticed and included in the list of problems. If the main effect of some action later in the plan is true before the action is executed, then that is noted as a possible place to shorten the plan (this is discussed in more detail in the next section).

Because of the way plans are encoded in SIPE, these are the only things that need be checked for determining if a MN node affects the remainder of a plan. It should be noted that this depends upon the fact that processes (actions) are assumed to work whenever their precondition is true and when all phantoms whose MAINSTEP slot points to the process are true. (All such necessary conditions should be encoded as either preconditions or goals in any case.) There is currently no check for loops in case the same error happens repeatedly with the same fix proposed each time. Various simple checks could easily be added if this were a problem.

4.1 Solution to the Truth Maintenance Problem

SIPE's solution to the truth maintenance problem is based on the efficiency of its deductive capability. Since it is assumed that processes work as expected whenever their precondition is true and all phantoms whose MAINSTEP slot points to the process are true, only the deduced effects need to be checked for their dependence on unexpected effects. (The execution monitor will solve problems having to do with preconditions and phantoms that are not true).

SIPE's deductive capability was designed to find a good balance between expressiveness and efficiency. While providing the power of many useful deductions, it nevertheless keeps deduction under control by severely restricting the deductions that can be made, and by having triggers to control the application of deductive operators. All deductions that can be made are performed at the time a node is inserted into the plan. Since deduction is not expensive, the truth maintenance problem is solved simply by redoing the deductions at each node in the plan after a MN node. Even this can be avoided in simple cases, because SIPE carries a list of changed predicates as it goes through the plan, and, if they all become true later in the plan (without any deduced effects changing in the interim), then the execution monitor need not look at the remainder of the plan (either for redoing deductions or for finding problems).

5. Replanning Actions

The six replanning actions described below, REINSTANTIATE, INSERT, INSERT-CONDITIONAL, RETRY, REDO, and POP-REDO, have all been implemented in SIPE. These actions

provide sufficient power to alter plans in a way that often retains much of the original plan. These are domain-independent actions, and the intention is to use them as a basis for domain-specific error-recovery operators in SIPE. They are also used in the general replanner. Both of these uses are described in more detail in the next section. In all actions below, the context argument merely specifies the context of the current plan.

The last three actions mentioned below all change the plan so that it will contain unsolved problems. The intention is that the plan will then later be given to the normal planning module of SIPE (possibly after a number of these replanning actions have changed the plan). The planner will then attempt to find a solution which solves all the problems that have been corrected in the plan. The planner automatically checks whether things it splices into the middle of the plan cause problems later in the plan so any solution found will be correct. (It does this when copying nodes down to the next lower level during planning.)

REINSTANTIATE (predicate node context)

The action attempts to instantiate a variable differently in order to make the given predicate true in the situation specified by the given node. This appears to be a commonly useful replanning action as it might correspond to using a different resource if something has gone wrong with the one originally employed in the plan, or deciding to return to the screw hopper for another screw rather than trying to find the one that has just been dropped.

This is done by looping through the arguments of the given predicate and, for each one, checking if there is another instantiation for it that will make the predicate true. This is cheap and efficient in SIPE since it merely involves removing the INSTAN constraint on the variable from the current context (and also from all variables constrained to be the same as this one), and then calling the normal matcher to determine if the predicate is now true (which will return possible instantiations). If new instantiations are found, the REINSTANTIATE action checks the remainder of the plan to see if any parts of it might be affected by the new instantiation. This is done by a routine similar to the problem detector described in Section 4 (in fact, the two share much of their code). REINSTANTIATE currently accepts new instantiations only if they cause

no new problems (see discussion below on tradeoffs). If all new instantiations are rejected, the old INSTAN constraint is simply replaced.

There are many tradeoffs in writing a replanning action such as this. There are also tradeoffs in deciding when to apply REINSTANTIATE as it exists, but these are discussed later in the paper. The implementation described above opts for reinstantiation only when it is likely to be the correct solution. For example, new instantiations could be accepted even though they cause problems if these problems are less severe than the problems entailed by keeping the old instantiation. Since SIPE has no way of comparing the difficulty of two sets of problems, REINSTANTIATE does not do this.

One could also expend more effort in finding new instantiations. As implemented, this replanning action will find reinstantiations when only one variable is changed. Some problems could be solved by reinstantiating a whole set of variables, but this would be more expensive and perhaps involve a search problem to decide which variables to include in the set. The decision to try only one variable was made because it appeared powerful enough to be useful. If the ability to reinstantiate sets of variables appeared useful, it would certainly be tractable to implement.

INSERT (node1 node2)

This action inserts the subplan beginning with node1 (which has been constructed) into the current plan after node2. All links between the new subplan and the old plan are correctly inserted. This is used as a subroutine by many of the actions below.

INSERT-CONDITIONAL (variable node context)

This action is not very interesting, but complements the unknown variable feature, which may be useful. It simply inserts a conditional around the given node that tests whether the given variable is known. If it is, the given node is executed next; otherwise a failure node is executed.

RETRY (node)

This replanning action is very simple. The given node is assumed to be a phantom node and it is changed to a goal node so that the planner will see it as unsolved.

REDO (pred node context)

This action creates a GOAL node whose goal is the given predicate. It then calls INSERT to put this new node after the given node in the plan. The planner will see the new node as an unsolved goal.

POP-REDO (node predicates context)

This is the most complicated of the replanning actions; it is used to remove a hierarchical wedge from the plan and, in some cases, replace it with a node at the lowest level. This is used both when a PRECONDITION node is no longer true and another action must be applied at a higher level, and when there may be a serendipitous unexpected effect. POP-REDO could also be used to find higher-level goals from which to replan when there are widespread problems causing the replanning to fail (this is not currently implemented).

In the case of redoing a precondition failure, it is easy to determine the wedge to be removed since PRECONDITION nodes are copied down from level to level. The top of the wedge to be removed is the node that was expanded to initially place the given PRECONDITION node (or one of its ancestors that is a PRECONDITION node) in the plan. Actually only the bottom of the wedge is spliced out of the plan, as planning will continue only from the lowest level. The subplan that is removed at the lowest level is replaced by a copy of the GOAL or CHOICEPROCESS node that was at the top of the wedge. (The INSERT replanning action is used for this.) This is seen as an unsolved goal by the planner, which automatically checks whether expansions of this node cause problems later in the plan.

There is one further complication involved. Various constraints may have been posted on the planning variables because of decisions made in the wedge of the plan that has been effectively removed. Because of SIPE's use of alternative contexts, this is easily solved. A context is a list of choicepoints, and constraints are posted relative to the choicepoint that forced them to be posted. Therefore, our problem is solved by removing from the current context all the choicepoints that occurred in the wedge of the plan that was effectively removed. This new context is given as the context argument to future planning actions, and no further action need be taken. This results in

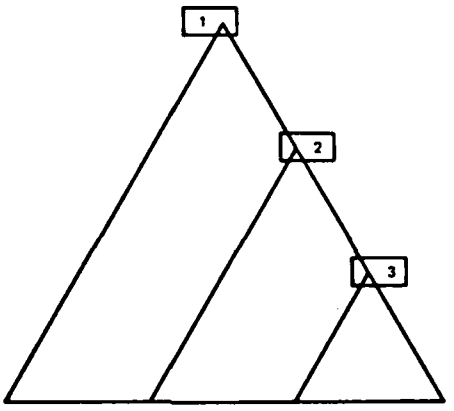


Figure 3
Hierarchical Wedges with a Common Last Action

ignoring exactly those constraints that should be ignored.

The case of serendipitous effects possibly shortening the plan is similar, except that, after the wedge is removed, no node is inserted. However, in this case it is nontrivial to decide which wedge to remove. There may be various wedges that are candidates, and, as with REINSTANTIATE, these candidates may cause problems later in the plan when they are removed. SIPE currently handles this case in the same way in handles the REINSTANTIATE case. Namely, it removes a wedge, checks if this causes any problems, and replaces the wedge if there are any problems. Thus serendipitous effects are taken advantage of only if doing so does not change the rest of the plan. This is a tradeoff like the one discussed previously.

SIPE also generates only one candidate wedge, and gives up taking advantage of the serendipitous effect if this one does not work. This candidate is generated by following ancestor links from the given node as long as some main effect of the candidate node is made true by one of the predicates in he list of given predicates. The candidate node found in this manner determines the candidate wedge. The wedge is rejected immediately unless all its main effects are true in the given list of predicates.

Figure 3 helps show the idea behind this selection process. Frequently, the last action at one level of a wedge will achieve the main effect of every level above that. The above selection process requires that all goals generated at a higher level than the candidate wedge be achieved before the wedge becomes a candidate, while goals generated at a lower level than the top of the candidate

wedge need not have been serendipitously achieved. Thus, for wedge 2 to be selected in Figure 3 , the serendipitous effects must include the main effects of the top of wedges 1 and 2, but need say nothing about the main effects of wedge 3. (It is assumed that, as long as the highest-level goal is achieved, we do not care about the lower-level goals that were necessary to bring this about.) The main effects of wedge 1 must be true, because they will be copied down to be effects of the top node of wedge 2 in the case when this is the node that achieves these effects.

6. Guiding the Replanning

The replanning actions of the previous section form the basis for a general replanning capability that has been implemented and for a language capable of specifying domain-specific error-recovery instructions that has been designed but not implemented. This section describes the automatic replanner and briefly outlines the error-recovery operators.

6.1 The General Replanner

The general replanner takes a list of problems from the problem recognizer described in Section 4 as well as possible serendipitous effects, and calls one or more of the replanning actions in Section 5 in an attempt to solve the problem. It first checks that a listed problem is still a problem since the REINSTANTIATE action may solve many problems at once.

If the problem is a purpose not being achieved, the system tries a REDO, which inserts the unachieved purpose as a GOAL node after the Mother-Nature node. If the problem is a previous phantom not being maintained, SIPE first tries REINSTANTIATE and, if that fails, it calls RETRY. The idea is that, if there is another object around with all the desired properties, then it would be easier to use that object than to reachieve the desired state with the original object. If a PROCESS node has an unknown variable as an argument, then INSERT-CONDITIONAL is called. If a future phantom is no longer true, then RETRY is called. As with maintaining phantoms, REINSTANTIATE may be more appropriate, but, in both cases, this depends entirely on the domain so the selection here is arbitrary. For preconditions that are not true, the general replanner first calls REINSTANTIATE and, if that fails, it calls POP-REDO. If parallel-postconditions are

not true, the general replanner constructs the appropriate parallel goals to reachieve them and calls INSERT to place them after the original JOIN node.

One cannot expect very impressive performance from a replanner that does not have domain-specific information for dealing with errors. For example, whether or not REINSTANTIATE is likely to succeed will be dependent on the domain. The automatic replanner makes reasonable guesses at what might be a good choice in the domains on which SIPE has been tested. Since it simply chooses a replanning action for each type of problem that is found, it is very simple and could easily be rewritten for different domains.

6.2 Error Recovery Language

We also plan to extend the operator description language so that instructions for handling foreseeable errors can be included in operators. The error recovery operators will be in the same syntax as all other SIPE operators, with some new additions made to this language as described below. The plots of these operators will include references to the replanning actions in Section 5. SIPE's ability to specify conditional plans in operators can be used to try a second replanning action only if the first fails.

The error-recovery operators will match their argument list to the arguments of the node being executed so original problem variables can be bound. There are two ways to invoke these operators, one for general operators that solve problems that have been recognized, and one for more specific operators that act directly on unexpected predicates. The latter ability seems attractive since it can avoid a lot of effort when there is good domain-dependent error-recovery information available. If one of these latter operators matches an unexpected predicate, it may be possible to simply apply the operator and assume that it will solve any problems caused by the unexpected predicate, thus shortcircuiting the normal problem detection mechanism.

The general operators will be applied after a MN node is added and problems have been found. Preconditions of these operators will be matched in the situation specified by the MN node. The general replanner will apply any general error-recovery operator that matches a given problem (the matching process is described below) instead of using its default replanning action.

Nodes in plots of regular operators will be able to specify an ERROR slot that gives names of error-recovery operators. The specific error-recovery instructions will be applied immediately after the input of an unexpected predicate (this will assume the predicate is problematical), but only when they are specified in the ERROR slot of the action being executed and match the predicate that was input (see below).

Like deductive operators, error-recovery operators will have a TRIGGER slot [5] to determine when they should be applied. The trigger will be a predicate (for specific operators) or a combination of keywords and predicates, where the keywords refer to the six types of problems. Specific operators are applied when their trigger matches the predicate that is input in the situation represented by the node being executed. General operators will have triggers that match when their keyword matches the problem being tried and any predicate in the trigger matches the appropriate predicates given in the problem.

7. Examples

This section presents two examples of SIPE actually monitoring the execution of a simple plan, and replanning when things do not go as expected. SIPE has been tested on larger and more complex problems than those presented here. These examples are simple to facilitate comprehension and to save space. Everything typed by the user is in boldface – nearly everything below is generated automatically by the system.

This first problem was constructed to show the successful use of the REINSTANTIATE replanning action. The problem is to get A on C in parallel with getting any blue block on any red block. In the initial world B1 and B2 are the only blue blocks (they are both on the table) and R1 and R2 are the only red blocks (R1 is on B1 and R2 is on the table and clear). Since A and C are both clear initially, the planner quickly finds a two-action plan of putting A on C in parallel with putting B2 on R2. While exectuing the moving of A to C, the SIPE is told that D has suddenly appeared on top of R2. It notices the problem in the parallel branch of the plan and the general replanner tries REINSTANTIATE, which succeeds. The original plan is retained in its entirety and B2 is placed on R1 instead of R2, thus achieving the original goal.

```
PROBLEM: PROB6
PARALLEL
BRANCH 1:
    GOALS: (ON A C);
BRANCH 2:
    GOALS: (ON BLUEBLOCK1 REDBLOCK1);
END PARALLEL
END PROBLEM

Plan being executed:
PLANHEAD: P0171
PLANHEAD: World model: (ON A B)(ON B TABLE)(ON C TABLE)(ON D TABLE)(ON E TABLE)
(ON R1 B1)(ON B1 TABLE)(ON B2 TABLE)(ON R2 TABLE)(CLEAR R1)(CLEAR R2)(CLEAR B2)
(CLEAR TABLE)(CLEAR A)(CLEAR D)(CLEAR E)(CLEAR C)

SPLIT: C0170
    Parallel branch:
    SPLIT: C0189
        Parallel branch:
        PHANTOM: P0194
        Goals: (CLEAR C);
        Mainstep: P0197;

        Parallel branch:
        PHANTOM: P0191
        Goals: (CLEAR A);
        Mainstep: P0197;
    JOIN: C0190

    PROCESS: P0197
    Action: PUTON.PRIM;
    Effects: (ON A C);
    Deduce: (CLEAR B), -(ON A B), -(ON A OBJECT3) OBJECT3 UNIVERSAL, -(CLEAR C);
    Mainstep: PURPOSE;

    Parallel branch:
    SPLIT: C0160
        Parallel branch:
        PHANTOM: P0165
        Goals: (CLEAR R2);
        Mainstep: P0168;

        Parallel branch:
        PHANTOM: P0162
        Goals: (CLEAR B2);
        Mainstep: P0168;
    JOIN: C0161

    PROCESS: P0168
```

Action: PUTON.PRIM;
Effects: (ON B2 R2);
Deduce: -(ON B2 TABLE), -(ON B2 OBJECT3) OBJECT3 UNIVERSAL, -(CLEAR R2);
Mainstep: PURPOSE;
JOIN: C0172

P0197 P0168
PICK ONE OF THE ABOVE NODES TO EXECUTE NEXT (? FOR HELP): P0197
Executing action P0197
Unexpected effect (? for help): **(ON D R2)**
Unexpected effect (? for help): **NIL**

Adding deduced predicate: -(ON D TABLE)
Adding deduced predicate: -(ON D OBJECT3) OBJECT3 UNIVERSAL
Adding deduced predicate: -(CLEAR R2)

Problem:
The following predicate negates predicate following it: -(CLEAR R2)
(CLEAR R2)
Causing purpose of following node not to be achieved.
EXECUTED: P0165
Effects: (CLEAR R2);
Mainstep: P0168;

Trying to reinstantiate to make this predicate true: (CLEAR R2)
matching condition: (CLEAR OBJECT1)
(collected 1 possibilities)
Adding INSTAN constraint: OBJECT1 BOUND TO R1
Success, new instantiation: OBJECT1 BOUND TO R1

New plan produced for continuing execution:
PLANHEAD: P0171
SPLIT: C0170
Parallel branch:
EXECUTED: P0197
Action: PUTON.PRIM;
Effects: (ON A C);
Deduce: (CLEAR B), -(ON A B), -(ON A OBJECT3) OBJECT3 UNIVERSAL, -(CLEAR C);

EXECUTED: P0350
Action: MOTHER.NATURE;
Effects: (ON D R2);
Deduce: -(ON D TABLE), -(ON D OBJECT3) OBJECT3 UNIVERSAL;

Parallel branch:
PROCESS: P0168
Action: PUTON.PRIM;
Effects: (ON B2 R1);
Deduce: -(ON B2 TABLE), -(ON B2 OBJECT3) OBJECT3 UNIVERSAL, -(CLEAR R1);

JOIN: C0172

Executing action P0168
Unexpected effect (? for help): **NIL**
Goal achieved.

The second problem, shown below, is the same as the first except that the red block is constrained not to be R1, which will cause REINSTANTIATE to fail. The original plan is the same and the unexpected situation is the same. This time SIPE tries REINSTANTIATE and it fails, so it calls RETRY, which causes the (CLEAR R2) phantom to be made into a goal. The planner solves this to produce a plan that puts D back on the table before B1 is placed on R2. The original plan is not shown below and the other plans are abbreviated by removing MAINSTEPS, deductions, and some phantoms.

PROBLEM: PROB7
PARALLEL
BRANCH 1:
    GOALS: (ON A C); BRANCH 2:
GOALS: (ON BLUEBLOCK1 REDBLOCK1); END PARALLEL

GOAL
ARGUMENTS: BLUEBLOCK2, REDBLOCK2 IS NOT R1;
GOALS: (ON BLUEBLOCK2 REDBLOCK2);
END PROBLEM

P0254 P0224
PICK ONE OF THE ABOVE NODES TO EXECUTE NEXT (? FOR HELP): P0254
Executing action P0254
Unexpected effect (? for help): **(ON D R2)**
Unexpected effect (? for help): **NIL**

Adding deduced predicate: -(ON D TABLE)
Adding deduced predicate: -(ON D OBJECT3) OBJECT3 UNIVERSAL
Adding deduced predicate: -(CLEAR R2)

Problem:
The following predicate negates predicate following it: -(CLEAR R2)
(CLEAR R2)
Causing purpose of following node not to be achieved.
EXECUTED: P0221
Effects: (CLEAR R2);

Trying to reinstantiate to make this predicate true:
(CLEAR R2)
matching condition: (CLEAR OBJECT1)

Condition fails.
P0221 being reset to GOAL for replanning.

initial problem:
PLANHEAD: P0227
PLANHEAD: World model: (ON D R2)(ON A C)(CLEAR B)(ON B TABLE)(ON C TABLE)
(ON E TABLE)(ON R1 B1)(ON B1 TABLE)(ON B2 TABLE)(ON R2 TABLE)(CLEAR R1)
(CLEAR B2)(CLEAR TABLE)(CLEAR A)(CLEAR D)(CLEAR E)

SPLIT: C0226
    Parallel branch:
    SPLIT: C0246
        Parallel branch:
        EXECUTED: P0251 Goals: (CLEAR C);
        Parallel branch:
        EXECUTED: P0248 Goals: (CLEAR A);
    JOIN: C0247
    EXECUTED: P0254
    Action: PUTON.PRIM;
    Effects: (ON A C);
    EXECUTED: P0366
    Action: MOTHER.NATURE;
    Effects: (ON D R2);
    Parallel branch:
    SPLIT: C0216
        Parallel branch:
        GOAL: P0221 Goals: (CLEAR R2);

        Parallel branch:
        EXECUTED: P0218 Effects: (CLEAR B2);
    JOIN: C0217

    PROCESS: P0224
    Action: PUTON.PRIM;
    Effects: (ON B2 R2);
JOIN: C0228

planner succeeds
Plan being executed:
SPLIT: C0448
    Parallel branch:
    SPLIT: C0483
        Parallel branch:
        EXECUTED: P0488 Goals: (CLEAR C);

        Parallel branch:
        EXECUTED: P0482 Goals: (CLEAR A);
    JOIN: C0485

```
EXECUTED: P0486 Action: PUTON.PRIM;
Effects: (ON A C);
EXECUTED: P0487 Action: MOTHER.NATURE;
Effects: (ON D R2);

Parallel branch:
SPLIT: C0446
    Parallel branch:
    EXECUTED: P0489 Goals: (CLEAR B2);

    Parallel branch:
    SPLIT: C0435
        Parallel branch:
        PHANTOM: P0440 Goals: (CLEAR TABLE);

        Parallel branch:
        PHANTOM: P0437 Goals: (CLEAR D);
    JOIN: C0436

    PROCESS: P0443 Action: PUTON.PRIM;
    Effects: (ON D TABLE);
    Deduce: (CLEAR R2), -(ON D R2), -(ON D OBJECT3) OBJECT3 UNIVERSAL;

    PHANTOM: P0450 Goals: (CLEAR R2);
JOIN: C0451

PROCESS: P0452 Action: PUTON.PRIM;
Effects: (ON B2 R2);
JOIN: C0453

PHANTOM: P0454 Goals: (ON B2 R2);

Executing action P0443
Unexpected effect (? for help): NIL
Executing action P0452
Unexpected effect (? for help): NIL
Goal achieved.
```

8. Comparison to other systems

There is very little previous work in this area since most domain-independent planning systems do not address the questions of execution monitoring and replanning (e.g., NONLIN [3] and DEVISER [4]). Hayes [1] and Sacerdoti [2] have addressed this problem. The approaches used in both these systems were considerably simpler and less powerful than SIPE. NOAH did not even allow the input of arbitrary predicates, so the general replanning problem never arose. It did

permit the user to specify that whole wedges had been executed at once, and did allow the node just executed to be planned for again if it failed. This essentially provides one limited replanning action that is useful only in very specific situations.

Hayes's system does allow the input of some information about unexpected situations. It is not clear what types of information can be provided, but it appears less general than the arbitrary predicates accepted by SIPE. The system's only replanning action is to delete part of the plan. This permits the planner to reachieve higher-level goals, but they must be the same higher-level goals that were already present in the plan. The system deletes everything that depended on any *effect of a decision* that is no longer valid. This will, in general, be wasteful since much of the plan may be unnecessarily removed. If only one of many effects of an action has failed, subplans depending on the effects that did not fail do not need to be deleted. SIPE would find problems with such subplans.

SIPE provides a much more powerful replanning capability than either of these systems. It allows input of arbitrary predicates, computes how these affect the rest of the plan (and only finds problems that really are problematical), and uses a large number of replanning actions (including REINSTANTIATE) to fix problems in ways that allow much of the original plan to be maintained.

9. Summary and Limitations

Given correct information about unexpected events, SIPE is able to determine how this affects the plan being executed, and, in many cases, is able to retain most of the original plan by making changes in it to avoid problems caused by these unexpected events. It also is capable of shortening the original plan when serendipitous events occur. It cannot solve difficult problems involving drastic changes to the expected state of the world, but it does handle many types of small errors that may happen frequently in a mobile robot domain. The execution-monitoring package does this without the necessity of planning originally to check for these errors.

The major contributions of this work center around taking advantage of the rich structure of SIPE's planner and its plans, and the development of a general set of replanning actions that are used as the basis of an automatic replanner and can be used as the basis of a language for specifying

domain-dependent error-recovery information. The replanner calls the standard planning system so that it can take advantage of the efficient frame-reasoning mechanisms in SIPE to quickly discover problems and potential fixes, and use the deductive capabilities to provide a reasonable solution to the truth maintenance problem. The fixes need involve only inserting new goals in the plan, since calling the planner as a subroutine will solve these goals in a manner that assures there will be no conflicts with the rest of the plan. SIPE's execution-monitoring capabilities make extensive use of the explicit representation of plan rationale in plans. The problem detector makes uses of the information encoded in MAINSTEP slots, phantoms, and preconditions to quickly find all the problems with a plan. Furthermore, it does not remove parts of the original plan unless the parts are actually problematical. SIPE's deductive capability is instrumental in the solution of the truth maintenance problem. The replanning actions make use of constraints, alternative contexts, and wedges in SIPE whenever they consider removing part of the plan.

Another important contribution is the development of a general set of replanning actions that will form the basis for a language capable of specifying error-recovery operators, and a general replanning capability that has been implemented using these actions. These actions provide sufficient power to alter plans in a way that often retains much of the original plan, (e.g., the REINSTANTIATE action). The general replanner attempts to solve all problems that are found. It is unlikely to be very successful without being tuned for particular domains. The design of the language for error-recovery operators allows for both operators that will handle very specific situations and operators that will give more general advice to the replanner.

The major limitations of this research result from the assumption of correct information about unexpected events. This avoids the hard problems of generating predicates from information provided by the sensors, deciding how much effort to expend checking facts that may be suspect, and modeling uncertain or unreliable sensors. These problems are all crucial to providing execution-monitoring capabilities to a mobile robot and must yet be addressed.

ACKNOWLEDGMENTS

Many people influenced the ideas expressed in this paper. Special thanks go to Michael Georgeff

for many enlightening discussions.

REFERENCES

1. Hayes, Philip J., "A Representation for Robot Plans", *Proceedings IJCAI-75*, Tbilisi, USSR, 1975, pp. 181-188.

2. Sacerdoti, E., *A Structure for Plans and Behavior*, Elsevier, North-Holland, New York, 1977.

3. Tate, A., "Generating Project Networks", *Proceedings IJCAI-77*, Cambridge, Massachusetts, 1977, pp. 888-893.

4. Vere, S., "Planning in Time: Windows and Durations for Activities and Goals", Jet Propulsion Lab, Pasadena, California, November 1981.

5. Wilkins, D., "Domain-independent Planning: Representation and Plan Generation", *Artificial Intelligence 22*, April 1984, pp. 269-301.

A Theory of Action for MultiAgent Planning

Michael Georgeff
Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, California 94025.

Abstract

A theory of action suitable for reasoning about events in multiagent or dynamically changing environments is presented. A device called a process model is used to represent the observable behavior of an agent in performing an action. This model is more general than previous models of action, allowing sequencing, selection, nondeterminism, iteration, and parallelism to be represented. It is shown how this model can be utilized in synthesizing plans and reasoning about concurrency. In particular, conditions are derived for determining whether or not concurrent actions are free from mutual interference. It is also indicated how this theory provides a basis for understanding and reasoning about action sentences in both natural and programming languages.

1. Introduction

If intelligent agents are to act rationally, they need to be able to reason about the effects of their actions. Furthermore, if the environment is dynamic, or includes other agents, they need to reason about the interaction between their actions and events in the environment, and must be able to synchronize their activities to achieve their goals.

Most previous work in action planning has assumed a single agent acting in a static world. In such cases, it is sufficient to represent actions as state change operators (e.g., [4], [9]). However, as in the study of the semantics of programming languages, the interpretation of actions as functions or relations breaks down when multiple actions can be performed concurrently. The problem is that, to reason about the effects of concurrent actions, we need to know *how* the actions are performed, not just their final effects.

Some attempts have recently been made to provide a better underlying theory for actions. McDermott [10] considers an action or event to be a set of sequences of states, and describes a temporal logic for reasoning about such actions and events. Allen [1] also considers an action to be a set of sequences of states, and specifies an action by giving the relationships among the intervals over which the action's conditions and effects are assumed to hold. However, while it is possible to state arbitrary properties of actions and events, it is not obvious how one could use these logics in synthesizing or verifying multiagent plans. [1]

In a previous paper [5], we proposed a method for forming synchronized plans that allowed multiple agents to achieve multiple goals, given a simple model of the manner in which the actions of one agent interact with those of other agents. In this paper, we propose a more general model of action, and show how it can be used in the synthesis or verification of multiagent plans and concurrent programs.

2. Process Models and Actions

Agents are machines or beings that act in a world. We distinguish between the internal workings of an agent and the external world that affects, and is affected by, that agent. All that can be observed is the external world. At any given instant, the world is in a particular *world state*, which can be described by specifying conditions that are true of that state.

Let us assume that the world develops through time by undergoing discrete changes of state. Some of these changes are caused by agents acting in the world; others occur "naturally," perhaps as a result of previous state changes. Actions and events are considered to be composed of primitive objects called *atomic transitions*. An atomic transition is a relation on the set of world states. Any sequence of states resulting from the application of some specified atomic transitions will be called an *event*. Note that we do not require that atomic transitions be deterministic, but we do require that they terminate.

An *action* is a class of events; viewed intuitively, those that result from the activity of some agent or agents in accomplishing some goal (including the achievement of desired conditions, the maintenance of desired invariants, the prevention of other events, etc.)

In carrying out or performing an action, an agent forces some sequence of atomic transitions in the world. For every action the agent is capable of performing, there will correspond some internal structure that specifies just how and under what conditions these atomic transitions are to be made.

[1] Allen [2] proposes a method for forming multiagent plans that is based on his representation of actions. However, he does not use the temporal logic directly, and actions are restricted to a particularly simple form (e.g., they do not include conditionals).

Usually we do not have access to this internal structure. However, since we are interested only in the *observable* behavior of the agent, we do not need to know the internal processes that govern the agent's actions. Thus, to reason about how the agent acts in the world and how these actions interact with events in the world, we need only an abstract model that explains the observable behavior of the agent.

We shall specify the class of possible and observable behaviors of an agent when it performs an action by means of a device called a *process model*. A process model consists of a number of internal states called *control points*. At any moment in time, *execution* can be at any one of these control points. Associated with each control point is a *correctness condition* that specifies the allowable states of the world at that control point.

The manner in which the device performs an action is described by a partial function, called the *process control function*, which, for a given control point and given atomic transition, determines the next control point. A process model can thus be viewed as a finite-state transition graph whose nodes are control points and whose arcs are labeled with atomic transitions.

A process model for an action stands in the same relationship to the internal workings of an agent and events in the external world as a grammar for a natural language bears to the internal linguistic structures of a speaker and the language that is spoken. That is, it models the observable behavior of the agent, without our claiming that the agent actually possesses or uses such a model to generate behaviors.

3. Formal Definition

A *process model* describes an action open to an agent. Formally, a process model is a seven-tuple $A = (S, F, C, \delta, P, c_I, c_F)$ where

- S is a set of *world states*
- $F : SXS$ is a set of *atomic transitions*
- C is a set of *control points*
- $\delta : C \times F \rightarrow C$ is a *process control function*
- $P : C \rightarrow 2^S$ associates subsets of S with each control point; values of this function are called *correctness conditions*
- $c_I \in C$ is the *initial control point*
- $c_F \in C$ is the *final control point*.

In general, δ is a partial function. If for a control point c and atomic transition tr, (c, tr) is in the domain of δ, we say that tr is *applicable* at c.

We are now in a position to define the execution of a process model. Let A be a process model as defined above. We first define a *state of execution* of A to be a pair (u, c), where [2] $c \in C$ and $u \in S^*$. We say that a state of execution $e_1 = (us_1, c_1)$ *directly generates* a state of execution $e_2 = (us_1s_2, c_2)$, denoted $e_1 \triangleright_A e_2$, if either

1. $\exists tr \,.\, \delta(c_1, tr) = c_2$ and $(s_1, s_2) \in tr$, or
2. $c_1 = c_2$

In (1) we say that the transition is effected by the agent executing A, while in (2) we say that the transition is effected by the *environment*.

We now define a restriction on the relation $\triangleright_A$. If, for e_1 and e_2 defined above, $e_1 \triangleright_A e_2$ and $s_2 \in P(c_2)$, we say that e_1 *successfully* generates e_2, denoted $e_1 \Rightarrow_A e_2$. If $s_2 \notin P(c_2)$, execution is said to *fail*.

Let $\Rightarrow_A^*$ denote the reflexive transitive closure of the relation $\Rightarrow_A$. Then the action generated by A, denoted α_A, is defined to be

$$\alpha_A = \{b \mid (s, c_I) \Rightarrow_A^* (b, c_F) \text{ and } s \in P(c_I)\}$$

Each element of α_A is called a *behavior* or *act* of A. The action α itself is the set of all behaviors resulting from the execution of A.

Viewed intuitively, the device works as follows. If it is at control point c_1 and the world is in a state s_1 satisfying the correctness condition $P(c_1)$, the device can pass to control point c_2 and the world to state s_2 as long as there exists an applicable atomic transition tr between states s_1 and s_2 and $\delta(c_1, tr) = c_2$. *Alternatively*, the device can stay at control point c_1 and some transition or event occur in the world (perhaps resulting from the action of some other agent). In either case, for the execution to be successful (not to fail), the new world state must satisfy the correctness condition at c_2, i.e., s_2 must be an element of $P(c_2)$.

In performing the action α, the device starts at control point c_I. The action terminates when the device reaches c_F. Given an initial state of the world s, various sequences of world states can be generated by the process model as it passes from the initial to the final control point. The set of all such sequences constitute the action itself.

This is the same general view of action as presented by Allen [1] and McDermott [10]. However, our theory differs in that it allows us to distinguish between transitions effected by the agent and those effected by the external world. This is particularly important in the synthesis and verification of multiagent plans and concurrent programs (e.g., [3]).

Note that we do not require that a state satisfying the correctness condition at a control point be in the domain of some atomic transition applicable at that control point. Thus, it is possible for the agent to arrive at an intermediate control point and not to be able to immediately effect a further transition. In such cases, the environment must change before the action can progress. This could occur, for example, if an agent nailing two boards together expected another to help by holding the boards. Only when the "holder" (who is part of the environment) has provided the necessary assistance (and moved the state of the world into

[2] S^* is the set of all finite sequences over S.

the domain of an applicable transition) can the "nailer" proceed with the action.

Neither do we require that an atomic transition performed by an agent always be successful i.e., the transition could sometimes leave the agent in a state that violated the current correctness condition. A process model that allowed such transitions could sometimes fail. In most cases, this is undesirable (though it may be unavoidable), and for the rest of the paper we will assume that this *cannot* happen. That is, we will assume that only the environment (or another agent) can cause an action to fail.

It should also be noted that the correctness conditions say nothing about termination — it may be that an action never reaches completion. This can be the case if the action is waiting for a condition to be satisfied by the environment (so that a transition can be effected), it loops forever, or the environment is *unfair* (i.e., does not give the action a chance to execute).

In many cases, we wish to model actions that proceed at an undetermined rate and fail if they are ever forced to suspend execution. For example, it is difficult to hit a golf ball if the environment is allowed to remove and replace the ball at arbitrary times during one's swing. Such uninterruptable actions require that, for any control point c, any state that satisfies the correctness condition at c also be in the domain of some atomic transition applicable at c.

4. Composition of Actions

A plan or program for an agent is a syntactic object consisting of primitive operations combined by constructions that represent sequencing, nondeterministic choice, iteration, forks and joins, etc. If we intend the denotations of such plans to be process models, we need some means of combining the latter in a way that reflects the composition operators in plans.

Of special interest, and indeed the motivation behind the model presented here, is the parallel-composition operator. We define this below.

Let $A_1 = \langle S, F_1, C_1, \delta_1, P_1, c_{I1}, c_{F1}\rangle$ and $A_2 = \langle S, F_2, C_2, \delta_2, P_2, c_{I2}, c_{F2}\rangle$ be two process models for actions α_1 and α_2, respectively. Then we define a process model representing the parallel composition of A_1 and A_2, denoted $A_1 \parallel A_2$, to be the process model $\langle S, F, C, \delta, P, c_I, c_F\rangle$, where

- $F = F_1 \cup F_2$
- $C = C_1 \times C_2$
- For all $c_1 \in C_1$, $c_2 \in C_2$ and tr in F_1, $\delta(\langle c_1, c_2\rangle, tr) = \langle \delta_1(c_1, tr), c_2\rangle$
- For all $c_1 \in C_1$, $c_2 \in C_2$ and tr in F_2, $\delta(\langle c_1, c_2\rangle, tr) = \langle c_1, \delta_2(c_2, tr)\rangle$
- For all $c_1 \in C_1$ and $c_2 \in C_2$, $P(\langle c_1, c_2\rangle) = P_1(c_1) \cap P_2(c_2)$
- $c_I = \langle c_{I1}, c_{I2}\rangle$
- $c_F = \langle c_{F1}, c_{F2}\rangle$

It is not difficult to show that the action α generated by $A_1 \parallel A_2$ is exactly $\{x \cap y \mid x \in \alpha_1 \text{ and } y \in \alpha_2\}$.

Note that the projection of δ onto C_1 and C_2 gives exactly the control function for the component process models. At any moment, each component is at one of its own control points; the pair of control points, taken together, represents the current control point of the parallel process.

Furthermore, the behaviors generated by these two processes running in parallel are also generated by each of them running separately. This means that any property of the behaviors of the independent processes can be used to determine the effect of the actions running in parallel. This is particularly important in providing a compositional logic for reasoning about such actions (see [3]).

The above model of parallel execution is an interleaving model. Such a model is adequate for representing almost all concurrent systems. The reason is that, in almost all cases, it is possible to decompose actions into more and more atomic transitions until the interleaving of transitions models the system's concurrency accurately. The nondeterministic form of the interleaving means that we make no assumption about the relative speeds of the actions. We can also define a parallel composition operator that is based on communication models of parallel action, in which communication acts are allowed to take place *simultaneously*. This, together with other composition operators, is described by me elsewhere [6].

5. Freedom from Interference

In plan synthesis and verification it is important to be able to determine whether or not concurrent actions interfere with one another. In the previous section we defined what it meant for two actions (strictly speaking, process models) to run in parallel. Now we have to determine whether execution of such a parallel process model could *fail* because of interaction between the two component processes.

Consider, then, two actions α and β generated by process models A and B, respectively. The process model corresponding to these actions being performed in parallel is $A \parallel B$. In analysing this model, however, we will view it in terms of its two component process models (i.e., A and B).

Assume that we are at control points c_1 in A and c_2 in B, and that tr is an atomic transition applicable at c_2. Clearly, if the process has not failed, the current world state must satisfy both $P(c_1)$ and $P(c_2)$. Now assume that process B continues by executing the atomic transition tr. This transition will take us to a new world state, while leaving us at the same control point within A. From A's point of view, this new state must still satisfy the condition $P(c_1)$. Thus, we can conclude that the transition tr executed at control point c_2 will not cause A to fail at c_1 if the following condition holds:

$$\forall s_1\ s_2\ .\ s_1 \in P(c_1) \cap P(c_2) \text{ and } \langle s_1, s_2\rangle \in tr \text{ implies } s_2 \in P(c_1)$$

We say that the transition tr at control point c_2 *does not interfere with* A if the above condition holds at all control points in A, i.e., for all correctness conditions associated with A.

We are now in a position to define freedom from interference. A set of process models $A_1, \ldots A_n$ is said to be *interference-free*[3] if the following holds for each process A_i: for all control points c in A_i and all transitions tr applicable at c and for all $j, j \neq i$, tr at c does not interfere with A_j.

Thus, if some set of actions is interference-free, none can be caused to fail because of interaction with the others. Of course, any of the actions could fail as a result of interaction with the environment.

From this it follows that, for ascertaining freedom from interference, it is sufficient to represent the functioning of a device by

1. A set of correctness conditions, and
2. A set of atomic transitions restricted to the correctness condition of the node from which they exit.

Knowledge of a process model's structure (i.e., the process control function), is unnecessary for this purpose. In a distributed system, this means that an agent need only make known the foregoing information to enable it to interact safely with other agents. We call such information a *reduced specification* of the action.

Let us consider the following example. Blocks A, B and C are currently on the floor. We wish to get blocks A and B on a table, and block C on a shelf, and have two agents, X and Y, for achieving this goal. Agent X has not got access to block B, but can place block A on the table and block C on the shelf. He therefore forms a plan for doing so. Agent Y cannot reach block A, but is happy to help with block B. Unfortunately, in doing so, he insists that the floor be clear of block C at the completion of his action.

The plans for agent X and Y are given below. The correctness conditions at each control point in the plans are shown in braces, "{" and "}". The "if" statement is assumed to be realized by two atomic transitions. The first of these is applicable when block C is on the floor, and results in block C being placed on the table. The second is applicable when block C is not on the floor, and does nothing (i.e., is a no-op). The process models corresponding to these plans should be self-evident.

Plan for agent X:

{(clear A) and (clear C)}
(puton A TABLE)
{(on A TABLE) and (clear C)}
(puton C SHELF)
{(on A TABLE) and (on C SHELF)}

[3]This definition of the notion "interference-free" generalizes to arbitrary transitions that used by Owicki and Gries[11] for verifying concurrent programs. Synchronization primitives have not been included explicitly, but can be handled by conditional atomic transitions [8].

Plan for agent Y:

{(clear P) and (clear C)}
(puton B TABLE)
{(on B TABLE) and (clear C)}
if (on C FLOOR) then
(PUTON C TABLE)
{(on B TABLE) and not (on C FLOOR)}

It is clear from the definition given above that these actions are interference-free. However, they interact in quite a complex manner. In some circumstances, agent Y *will* put block C on the table, which would seem to suggest interference. Nevertheless, interference freedom is assured because the only time that Y can do this is when it does not matter, i.e., before X has attempted to put C on the shelf. Note that if the test and action parts of the "if" statement were separate atomic transitions, rather than a single one, then the actions would not be free from interference.

6. General Reasoning about Actions

So far we have been interested solely in reasoning about possible interference among actions. For many applications, we may wish to reason more generally about actions. One way to do this is to construct a logic suitable for reasoning about process models and the behaviors they generate. That is, we let process models serve as interpretations for plans or programs in the logic. An interesting compositional temporal logic has been developed by Barringer et al [3]. Because it is compositional, process models provide a natural interpretation for the logic.

One may well ask what role process models play, given that the only observables are sequences of world states and that a suitable temporal logic, per se, is adequate for describing such sequences. However, in *planning* to achieve some goal, or *synthesizing* a program, we are required to do more than just describe an action in an arbitrary way — we must somehow form an object that allows us to choose our *next* action (or atomic transition) purely on the basis of the current execution state, without any need for further reasoning.

We could do this by producing a temporal assertion about the action from which, at any moment of time, we could directly ascertain the next operation to perform (e.g., a formula consisting of appropriately nested "next" operators). Thus, in a pure temporal logic formalism, plan synthesis would require finding an approriately structured temporal formula from which it was possible to deduce satisfaction of the plan specification. However, instead of viewing planning syntactically (i.e., as finding temporal formulas with certain structural properties), it is preferable, and more intuitive, to have a model (such as a process model) that explicitly represents the denotation of a plan or program (see [6]).

Process models serve other purposes also. For example, interference freedom is easily determined, given a process model, but it is less clear how this could be achieved ef-

ficiently, given a general specification in a temporal logic. Even so, one would need to construct an appropriate process model first (or its syntactic equivalent in a temporal logic), as the implementation of the specifications might make it necessary to place additional constraints upon the plan.

In combination with a temporal logic such as suggested above, the proposed theory of action provides a semantic basis for commonsense reasoning and natural-language understanding. Process models are more general than previously proposed models (e.g., [7]), particularly in the way they allow parallel composition. They can represent most actions describable in English, including those that are problematic when actions are viewed as simple state-change operators, such as "walking to the store while juggling three balls" [1], "running around a track three times" [10], or "balancing a ball" (which requires a very complex process model despite the apparent simplicity of its temporal specification). The theory also allows one to make sense of such notions as "sameness" of actions, incomplete actions (like an interrupted painting of a picture) and other important issues in natural-language understanding and commonsense reasoning.

Process models are also suitable for representing most programming constructs, including sequencing, nondeterministic choice (including conditionals) and iteration. Parallelism can also be represented, using either an interleaving model, as described in section 4, or a communication model. The model used by Owicki and Gries [11] to describe the semantics of concurrent programs can be considered a special case of that proposed herein.

7. Conclusions

A nascent theory of action suitable for reasoning about interaction in multiagent or dynamically changing environments has been presented. More general than previous theories of action, this theory provides a semantics for action statements in both natural and programming languages.

The theory is based on a device called a process model, which is used to represent the observable behavior of an agent in performing an action. It was shown how this model can be utilized for reasoning about multiagent plans and concurrent programs. In particular, a parallel-composition operator was defined, and conditions for determining freedom from interference for concurrent actions were derived. The use of process models as interpretations of temporal logics suitable for reasoning about plans and programs was also indicated.

REFERENCES

[1] Allen, J. F., "A General Model of Action and Time," Comp Sci Report TR 97, University of Rochester (1981).

[2] Allen, J.F., "Maintaining Knowledge about Temporal Intervals," *Comm. ACM*, Vol 26, pp. 832-843 (1983).

[3] Barringer, H., Kuiper, R., and Pnueli, A., "Now You May Compose Temporal Logic Specifications" (1984).

[4] Fikes, R.E., Hart, P.E., and Nilsson, N.J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, Vol 2, pp. 189-208 (1971).

[5] Georgeff, M.P., "Communication and Interaction in Multiagent Planning," *Proc. AAAI-83*, pp. 125-129 (1983).

[6] Georgeff, M.P., "A Theory of Plans and Actions," SRI AIC Technical Report, Menlo Park, California (1984).

[7] Hendrix, G.G., "Modeling Simultaneous Actions and Continuous Processes," *Artificial Intelligence*, Vol 4 (1973).

[8] Lamport, L., and Schneider, F.B., "The "Hoare Logic" of CSP, and All That", ACM Transactions on Programming Languages, Vol 6, pp 281-296 (1984).

[9] McCarthy, J., "Programs with Common Sense," in *Semantic Information Processing* M. Minsky ed. (MIT Press, Cambridge, Massachusetts) (1968).

[10] McDermott, D., "A Temporal Logic for Reasoning about Plans and Processes," Comp. Sci. Research Report 196, Yale University (1981).

[11] Owicki, S. and Gries, D., "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Comm. ACM*, Vol 19, pp 279-285 (1976).

END

FILMED

2-85

DTIC